ENGENHARIA DE SOFTWARE

AULA 5

Prof. Alex Mateus Porn

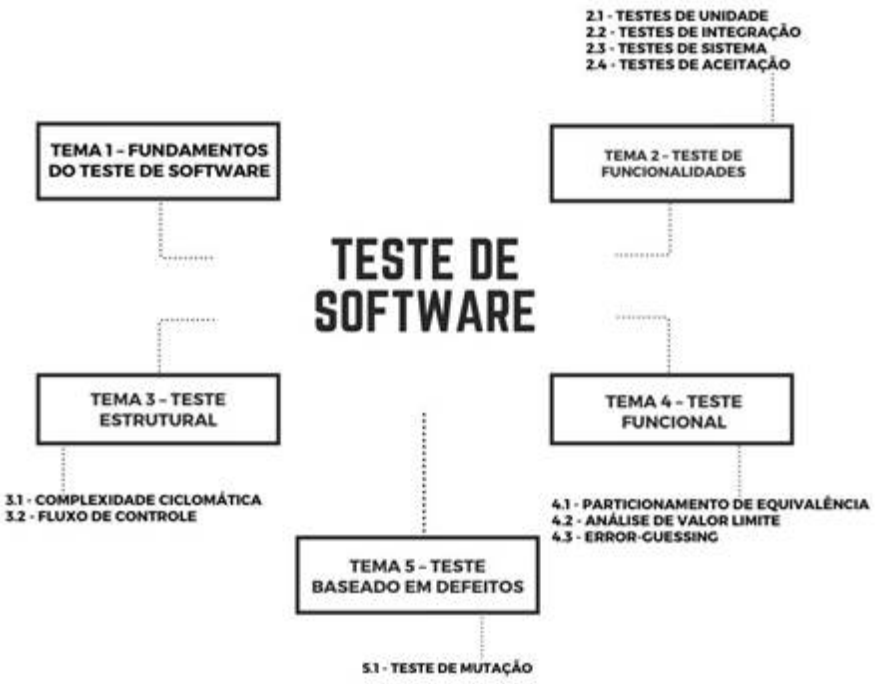
CONVERSA INICIAL

Olá! Esta é mais uma aula de Engenharia de Software. Nela, estudaremos os conceitos e as aplicações de uma das fases do desenvolvimento mais importantes, o teste de *software*. Temos como objetivo desta aula compreender os conceitos do teste de *software*, para que possamos identificar possíveis defeitos cometidos no desenvolvimento do programa, por exemplo ao gerar casos de teste adequados com potencial de revelar esses defeitos.

Três técnicas de teste de *software* são propostas na literatura – funcional, estrutural e baseada em defeitos. Cada uma dessas técnicas apresenta diversos critérios de teste, que em sua maioria propõem atingir o mesmo objetivo: revelar defeitos no *software* que está sendo testado. O que distingue cada uma dessas três técnicas é a fonte utilizada para definir os requisitos de teste. De acordo com Delamaro, Maldonado e Jino (2007), cada critério procura explorar determinados tipos de defeitos, estabelecendo requisitos de teste para os quais valores específicos do domínio de entrada do programa devem ser definidos com o intuito de exercitá-los.

Abordaremos cada uma destas três técnicas, como os seus principais critérios de teste. Para cada um dos critérios estudados, com fins metodológicos, aplicaremos o critério de teste em um código-fonte exemplo.

Ao longo desta aula, serão trabalhados os seguintes conteúdos:



TEMA 1 – FUNDAMENTOS DO TESTE DE SOFTWARE

Podemos iniciar nossa aula com uma afirmação de Roger S. Pressman sobre o teste de *software*, na qual afirma que o teste é um processo individualista, e o número de tipos diferentes de testes varia tanto quanto as diferentes abordagens de desenvolvimento de *software* (Pressman, 2011).

Porém, partindo da afirmação de Pressman sobre a existência de uma vasta variedade de tipos diferentes de teste, alguns conceitos são padronizados para qualquer técnica ou critério de teste empregado no processo. De acordo com Wazlawick (2013), alguns termos poderiam ser considerados sinônimos, mas na literatura de teste apresentam significados bastante precisos, e as suas compreensões auxiliam na avaliação do processo de teste empregado:

- **Erro (*error*):** diferença detectada entre o resultado obtido de um processo computacional e o resultado correto ou esperado.
- **Defeito (*fault*):** linha de código, bloco ou conjunto de dados incorretos que provocam um erro.
- **Falha (*failure*):** não funcionamento do *software*, possivelmente provocado por um defeito, mas com outras causas possíveis.
- **Engano (*mistake*):** ação que produz um defeito no *software*.

Cabe destacar, com base na interpretação de Delamaro, Maldonado e Jino (2007) dos conceitos de engano, defeito, erro e falha do padrão IEEE 610.12 (1990), que:

- O engano caracteriza-se como uma ação equivocada de um conceito específico do domínio, geralmente causado por humanos.
- A existência de um defeito em um programa leva à ocorrência de um erro, desde que o defeito seja executado.
- O erro caracteriza-se por um estado inconsistente ou inesperado devido à execução de um defeito.
- Este estado inconsistente ou inesperado pode ocasionar uma falha.

Com a intenção de exemplificar esses conceitos na prática, vamos propor o seguinte exemplo de um programa computacional: nosso programa recebe como entrada dois valores inteiros (“x” e “y”) e deve computar o valor de x^y se x e y forem maior do que 1. Portanto, temos os seguintes requisitos:

1. Solicitar dois números inteiros ao usuário do programa.

2. Os números informados pelo usuário devem ser maiores do que 1.

3. Se os números informados forem menores ou iguais a 1, o programa deve retornar uma mensagem informando ao usuário que os números digitados estão incorretos.

4. Se os números digitados forem maiores do que 1, o programa deve computar x^y.

5. Após computar x^y, o programa deve perguntar ao usuário se ele deseja realizar um novo cálculo.

6. Se a resposta do usuário for *Sim*, o programa deverá solicitar mais dois valores.

7. Se a resposta do usuário for *Não*, o programa deverá ser encerrado.

Com base nos requisitos apresentados, podemos ter a seguinte versão do programa, conforme apresentado na Figura 1. Consideraremos nesse exemplo o código-fonte escrito em Portugol, para não precisarmos nos referenciar a situações específicas de cada linguagem de programação.

Figura 1 – Programa para computar x^y

```
1  programa
2  {
3      funcao inicio() {
4          inteiro x, y, result
5          escreva("Digite o valor da base e do expoente na sequência")
6          leia(x, y)
7          enquanto((x <= 1) ou (y <= 1)) {
8              escreva("Valores incorretos, digite novamente")
9              escreva("Digite o valor da base e do expoente na sequência")
10             leia(x, y)
11         }
12         result = x;
13         enquanto(y > 1) {
14             result = result * x
15             y = y - 1
16         }
17         escreva("Resultado = ", result)
18     }
19 }
```

Tomando como base o programa da Figura 1 como escrito de forma correta, poderíamos fazer as seguintes análises em relação aos conceitos de engano, defeito, erro e falha:

- **Engano:**
  - Linha 7: trocar os sinais de "<=" por somente "<";
  - Linha 15: trocar o sinal na operação "y – 1" por "y + 1".
- **Defeito:**
  - Ao executar o programa, se o usuário digitar o número 1 para a base e qualquer número positivo para o expoente, ou vice-versa, ao processar a linha 7, o programa computará o cálculo e identificará um defeito incompatível com o requisito número 3.
- **Erro:**
  - O defeito executado na linha 7 produzirá o erro de não gerar a mensagem conforme o requisito número 3 e computar o cálculo incompatível com o requisito número 4.
- **Falha:**
  - Ao executar a linha 15, será identificado o defeito de uso de um operador matemático incorreto, que produzirá um erro na geração do cálculo, ocasionando uma falha conhecida como *loop infinito*, na qual o programa não é finalizado e não apresenta o resultado esperado pelo usuário.

Na concepção de Delamaro, Maldonado e Jino (2007), para os erros serem descobertos antes de o *software* ser liberado para utilização, existe uma série de atividades, coletivamente chamadas de *verificação, validação e teste*, com a finalidade de garantir que tanto o modelo pelo qual o *software* está sendo construído quanto o produto em si estejam em conformidade com o especificado.

Com base nessa compreensão, Wazlawick (2013) conceitua os termos *verificação, validação e teste* como:

- **Verificação:** consiste em analisar o *software* para ver se ele está sendo construído de acordo com o que foi especificado.
- **Validação:** significa analisar o *software* construído para ver se ele atende às verdadeiras necessidades dos interessados.
- **Teste:** designa a atividade que permite realizar a verificação e a validação do *software*.

Em sua maioria, o teste baseia-se em definir um conjunto de dados de teste de entrada de um programa a ser testado juntamente com os resultados esperados de cada dado, executar o programa com esse conjunto de dados de teste e, em seguida, comparar os resultados obtidos com os resultados esperados.

Portanto, conforme destaca Delamaro, Maldonado e Jino (2007), dados de teste e casos de teste são caracterizados como:

- **Dados de teste:** é um elemento do domínio de entrada de um programa.
    - Para o programa que computa x^y, com o domínio de entrada (x, y) maiores do que 1, um dado de teste pode ser (2, 5).
- **Casos de teste:** é um par formado por um dado de teste mais o resultado esperado para a execução do programa com aquele dado de teste.
    - Para o programa que computa x^y, com o domínio de entrada (x, y) maiores do que 1, o caso de teste do dado de teste (2, 5) seria ((2, 5), 10).

Geralmente, o conjunto de dados de entrada de um programa em teste é obtido conforme o seu domínio de entrada. Nesse contexto, Delamaro, Maldonado e Jino (2007, p. 2) destacam que "o domínio de entrada de um programa P, denotado por D(P), é o conjunto de todos os possíveis valores que podem ser utilizados para executar P."

Tomando como base o nosso exemplo do programa que computa x^y, para todos x e y maiores do que 1, o domínio de entrada desse programa é formado por todos os possíveis pares (x, y) de números inteiros maiores que 1. Conforme Delamaro, Maldonado e Jino (2007), estabelecendo-se o domínio de entrada de um programa, é possível determinar o seu domínio de saída, que em nosso caso é o

conjunto de todos os possíveis resultados produzidos pelo programa, composto pelo conjunto de números inteiros e mensagens de erro produzidos pelo programa.

Utilizar o domínio de entrada de um programa para testá-lo pode tornar-se impraticável, pois, como no nosso exemplo, esse domínio é infinito. Conforme destaca Delamaro, Maldonado e Jino (2007, p. 4),

[...] para o programa que computa xy, com o domínio de entrada formado pelos números inteiros positivos maiores do que 1, isso produz uma cardinalidade de 2n * 2n, onde n é o número de bits usado para representar um número inteiro. Em uma arquitetura com 32 bits isso representa 264 = 18.446.744.073.709.551.616. Se cada caso de teste pudesse ser executado em 1 milissegundo, precisaríamos de 5.849.424 séculos para executá-los todos.

Nesse contexto, faz-se necessário que utilizemos subconjuntos do domínio de entrada para executar um programa em teste, que tenham alta probabilidade de revelar um defeito caso exista. Para isso, Delamaro, Maldonado e Jino (2007) apresentam duas formas de selecionar subconjuntos de casos de teste:

- **Teste aleatório:** os subconjuntos de casos de teste são selecionados aleatoriamente, de modo que, probabilisticamente, tenha-se uma boa chance de que esses subconjuntos estejam todos representados no domínio de entrada;
- **Teste de partição:** procura-se estabelecer quais são os subconjuntos a serem utilizados e, então, selecionam-se os casos de teste em cada subconjunto.

Cabe destacar que, em relação ao teste de partição, são estabelecidas certas regras para identificar quando dados de teste devem estar no mesmo subconjunto ou não. Em geral, são definidos requisitos de teste, como executar determinadas estruturas do programa separadamente. Os dados de teste que satisfizerem esse requisito pertencem ao mesmo subconjunto (Delamaro; Maldonado; Jino, 2007).

TEMA 2 – TESTE DE FUNCIONALIDADE

Segundo Wazlawick (2013), os testes de funcionalidade têm como objetivo basicamente verificar e validar se as funções implementadas no *software* estão corretas nos seus diversos níveis. Nesse

método de teste, encontram-se os testes de unidade, integração, sistema e aceitação, os quais veremos na sequência.

2.1 TESTES DE UNIDADE

Conforme Wazlawick (2013), os testes de unidade são os mais básicos e costumam consistir em verificar se um componente individual do *software* (unidade) foi implementado corretamente.

Como um exemplo de teste de unidade, podemos verificar se uma unidade do nosso programa que computa x^y foi corretamente implementada. Consideraremos neste exemplo como uma unidade do programa o laço de repetição implementado entre as linhas 13 e 16 do código apresentado na Figura 1. Sabemos que a especificação dessa unidade estabelece que ela deve calcular x^y de acordo com os valores de x e y passados pelo usuário, mas, antes disso, o programa deve verificar se os valores são maiores do que 1, pois, caso não sejam, deve ser solicitado que o usuário informe novos valores para x e y. Casos de teste de unidade poderiam ser escritos da seguinte forma:

- Inserir um valor para x e para y menores do que 1 e verificar se o programa solicita novos valores ao usuário.
- Inserir um valor para x maior do que 1 e para y menor do que 1 e verificar se o programa solicita novos valores ao usuário.
- Inserir um valor para x menor do que 1 e para y maior do que 1 e verificar se o programa solicita novos valores ao usuário.
- Inserir um valor para x e para y igual a 1 e verificar se o programa solicita novos valores ao usuário.
- Inserir um valor para x e para y maior do que 1 e verificar se o programa realiza o cálculo corretamente.

2.2 TESTES DE INTEGRAÇÃO

De acordo com Wazlawick (2013), testes de integração são feitos quando unidades estão prontas, são testadas isoladamente e precisam ser integradas para gerar uma nova versão do sistema. Entre as estratégias de integração, Wazlawick destaca as seguintes:

- **Integração *big-bang*:** consiste em construir as diferentes classes ou componentes separadamente e depois as integrar no final.
- **Integração *bottom-up*:** consiste em integrar inicialmente os módulos de mais baixo nível, ou seja, aqueles que não dependem de nenhum outro, e depois ir integrando os módulos de nível imediatamente mais alto.
- **Integração *top-down*:** consiste em integrar inicialmente os módulos de nível mais alto, deixando os mais básicos para o fim.
- **Integração sanduíche:** consiste em integrar os módulos de nível mais alto da forma *top-down* e os de nível mais baixo da forma *bottom-up*.

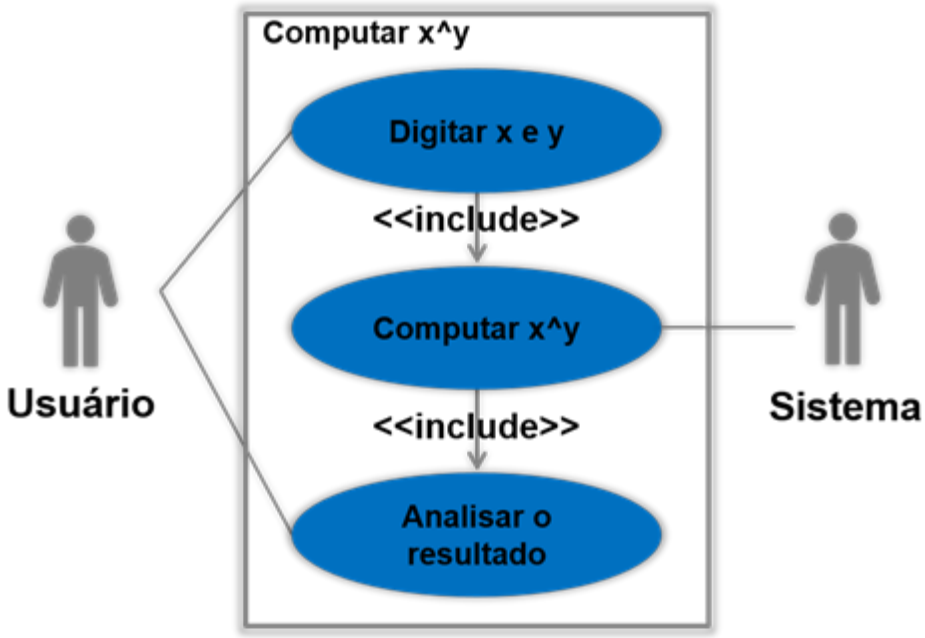
Para simplificar a compreensão desse tipo de teste, vamos considerar como duas unidades do nosso programa que computa x^y a unidade do exemplo do teste de unidade e o laço de repetição entre as linhas 7 e 11 da Figura 1, que verifica se os valores informados pelo usuário para x e y são maiores do que 1. No teste de unidade, verificamos cada uma dessas duas unidades independentemente. Como o nosso programa é composto somente por elas, nosso teste de integração consiste em analisar se o programa computa o cálculo corretamente com base nos valores informados pelo usuário ou se ele solicita novos valores caso x ou y seja menor ou igual a 1. Os casos de teste podem ser os mesmos utilizados no exemplo do teste de unidade.

2.3 TESTES DE SISTEMA

Conforme explicitado por Wazlawick (2013), o teste de sistema verifica se a atual versão do sistema permite executar processos ou casos de uso completos do ponto de vista do usuário e é capaz de obter os resultados esperados. Se cada uma das operações do sistema já estiver testada e integrada corretamente, então se deve verificar se o fluxo principal do caso de uso pode ser executado corretamente, obtendo os resultados desejados, bem como os fluxos alternativos.

Simplificando a nossa abordagem, a Figura 2 apresenta o diagrama de casos de uso do nosso programa de exemplo que computa x^y.

Figura 2 – Diagrama de casos de uso do sistema de exemplo que computa x^y



- **Caso de uso:** digitar x e y.
  - **Entrada:** o usuário informa os valores de x e y.
  - **Alternativo:** o usuário informa um valor incompatível, x ou y < 1.
    - O sistema solicita novos valores ao usuário e o caso de uso é reiniciado.
    - **Caso de uso:** computar x^y.
  - O programa calcula automaticamente x^y com base nos valores informados pelo usuário.
- **Caso de uso:** analisar o resultado.
  - **Saída:** o usuário analisa o resultado obtido.

Conforme a Figura 2, o fluxo principal do sistema consiste em o usuário digitar os dois valores solicitados pelo programa, x e y, realizar o cálculo automaticamente e apresentar o resultado ao usuário. Como fluxo alternativo, caso o usuário informe um valor para x ou y incompatível com o requisito do sistema, ou seja, números inteiros menores do que 1, o programa deve solicitar novos valores ao usuário.

Como nos testes anteriores, unidade e integração, testamos de modo independente cada unidade e posteriormente as suas integrações. De acordo com Wazlawick (2013), considera-se que o teste de sistema somente é executado em uma versão do sistema em que todas as unidades e os componentes recém-integrados já foram testados. Ainda segundo Wazlawick, caso muitos erros sejam encontrados durante esse teste, o usual é abortar o processo e refazer, ou mesmo replanejar, os testes de unidade e integração, para que uma versão suficientemente estável do sistema seja produzida e possa ser testada.

2.4 TESTES DE ACEITAÇÃO

O teste de aceitação, conforme o nome sugere, é um tipo de teste realizado pelo cliente ou usuários do sistema que consiste justamente na aceitação da aplicação desenvolvida. De acordo com Wazlawick (2013), o teste de aceitação costuma ser realizado utilizando-se a interface final do sistema. Ele pode ser planejado e executado exatamente como o teste de sistema, mas a diferença é que é realizado pelo usuário final ou cliente, e não pela equipe de desenvolvimento.

Conforme afirma Wazlawick (2013, p. 296):

Enquanto o teste de sistema faz a verificação do sistema, o teste de aceitação faz a sua validação. O teste de aceitação tem como objetivo principal, portanto, a validação do software quanto aos requisitos, e não a verificação de defeitos. Ao final do teste de aceitação, o cliente poderá aprovar a versão do sistema testado ou solicitar modificações.

TEMA 3 – TESTE ESTRUTURAL

O teste estrutural é uma técnica também conhecida como *testes de caixa branca*, pois todos os testes são executados com conhecimento do código-fonte. Nesse contexto, Wazlawick (2013) destaca que o teste estrutural é capaz de detectar uma quantidade substancial de possíveis erros pela garantia de ter executado pelo menos uma vez todos os comandos e condições do programa.

Nessa técnica de teste, podemos destacar, dentre os possíveis critérios de teste, os critérios baseados na complexidade e no fluxo de controle, conforme abordaremos na sequência.

3.1 CRITÉRIOS BASEADOS NA COMPLEXIDADE

De acordo com Delamaro, Maldonado e Jino (2007), os critérios baseados na complexidade utilizam informações sobre a complexidade do programa para derivar requisitos de *software*. Dois critérios bastante conhecidos são a complexidade ciclomática e os caminhos linearmente independentes.

3.1.1 COMPLEXIDADE CICLOMÁTICA

É uma métrica de *software* que proporciona uma medida quantitativa da complexidade lógica de um programa (Delamaro; Maldonado; Jino, 2007). Conforme abordado por Wazlawick (2013), se *n* é o número de estruturas de seleção e repetição no programa, então a complexidade ciclomática é *n*+1.

Contam-se as seguintes estruturas:

- IF-THEN: 1 ponto.
- IF-THEN-ELSE: 1 ponto.
- CASE: 1 ponto para cada opção, exceto OTHERWISE.
- FOR: 1 ponto.
- REPEAT: 1 ponto.
- OR ou AND na condição de qualquer das estruturas acima: acrescenta-se 1 ponto para cada OR ou AND (ou qualquer outro operador lógico binário, se a linguagem suportar, como XOR ou IMPLIES).
- NOT: não conta.
- Chamada de sub-rotina (inclusive recursiva): não conta.
- Estruturas de seleção e repetição em sub-rotinas ou programas chamados: não conta.

Com base na forma de pontuação apresentada acima, assume-se em geral, conforme destaca Wazlawick (2013, p. 300) que:

- Programas com complexidade ciclomática menor ou igual a 10 são simples e fáceis de testar;
- Programas com complexidade ciclomática de 11 a 20, são de médio risco em relação ao teste;
- Programas com complexidade ciclomática de 21 a 50, são de alto risco;
- Programas com complexidade ciclomática acima de 50 não são testáveis. O termo “programa” refere-se a um bloco de código único.

Analisando o código-fonte do nosso programa de exemplo apresentado na Figura 1, para calcular a sua complexidade ciclomática, temos a seguinte estrutura:

- Dois laços de repetição “enquanto” = 2 pontos.
- Um operador “ou” = 1 ponto.
- **Complexidade ciclomática = 3 pontos.**

Tendo como base a análise apresentada por Wazlawick, nosso sistema de exemplo que computa x^y possui baixa complexidade (menor que 10), sendo, portanto, considerado fácil de testar.

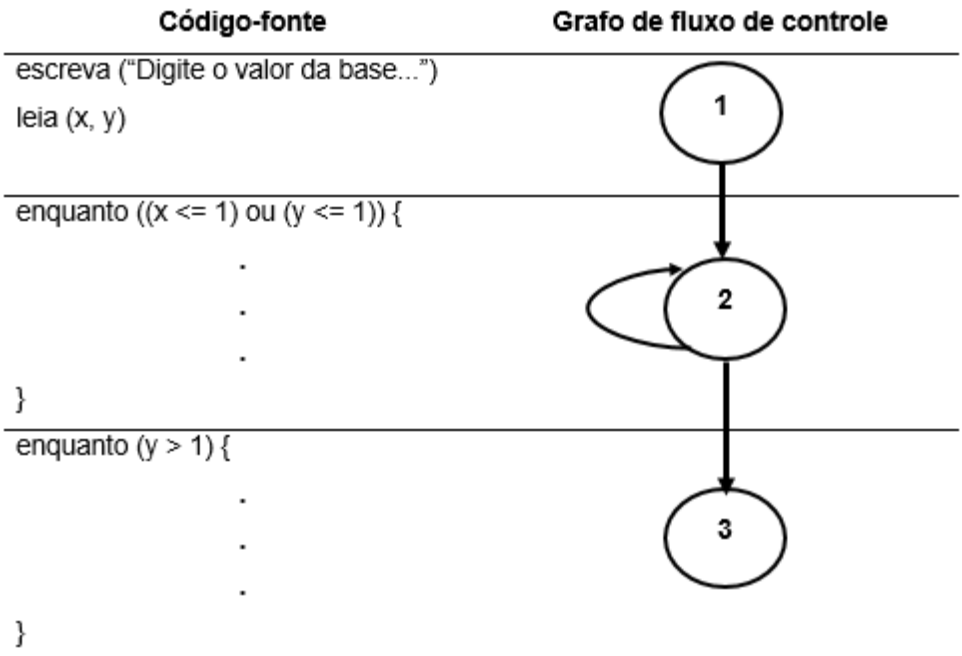
3.1.2 CAMINHOS LINEARMENTE INDEPENDENTES

O critério de teste caminhos linearmente independentes, conforme destacado por Delamaro, Maldonado e Jino (2007), é qualquer caminho do programa que introduza pelo menos um novo conjunto de instruções de processamento ou uma nova condição.

Para analisarmos os caminhos de um programa, podemos representá-lo pelo grafo de fluxo de controle (GFC). De acordo com a abordagem de Wazlawick (2013), o GFC é obtido colocando-se todos os comandos em nós e os fluxos de controle em arestas. Comandos em sequência podem ser colocados em um único nó, e estruturas de seleção e repetição devem ser representadas por meio de nós distintos com arestas que indiquem a decisão e a repetição, quando for o caso.

Com base nessa abordagem, na Figura 3 podemos visualizar a representação do GFC do nosso programa de exemplo da Figura 1.

Figura 3 – Grafo de fluxo de controle do programa que computa x^y.



Conforme destaca Wazlawick (2013), o valor da complexidade ciclomática indica o número máximo de caminhos necessários para exercitar todos os comandos do programa. No caso do exemplo apresentado na Figura 3, que corresponde ao nosso programa da Figura 1 que computa x^y, nós temos uma complexidade ciclomática de três pontos, que equivale ao número máximo de execuções até a finalização do nosso programa.

Porém, ainda de acordo com Wazlawick (2013), a execução de todos os comandos (nós) não vai necessariamente testar os valores verdadeiro e falso de todas as condições possíveis. Por exemplo, é possível passar por todos os nós do GFC da Figura 3 sem nunca passar pela aresta que vai do nó 2 a ele mesmo. Portanto, uma abordagem mais adequada para o teste é exercitar não apenas todos os nodos, mas todas as arestas do GFC. Baseando-se em Wazlawick (2013), isso é feito pela determinação dos caminhos independentes do grafo, que são possíveis navegações do início ao fim do grafo.

3.2 CRITÉRIOS BASEADOS NO FLUXO DE CONTROLE

Os critérios baseados no fluxo de controle fazem uso do Grafo de Fluxo de Controle de modo similar ao critério de caminhos linearmente independentes que estudamos no tópico anterior. Parafraseando Delamaro, Maldonado e Jino (2007, p. 56), esses critérios utilizam apenas características de controle da execução do programa, como comandos ou desvios, para determinar quais estruturas são necessárias. Entre os critérios mais conhecidos dessa categoria incluem-se:

- **Todos-Nós**: exige que a execução do programa passe, ao menos uma vez, em cada vértice do GFC, ou seja, que cada comando do programa seja executado pelo menos uma vez.
- **Exemplo**: com base na Figura 3, o critério Todos-Nós exige que sejam criados casos de teste que executem ao menos uma vez os nodos 1, 2 e 3.
- **Todas-Arestas:** requer que cada aresta do grafo, isto é, cada desvio de fluxo de controle do programa, seja exercitada pelo menos uma vez.
- **Exemplo:** com base na Figura 3, o critério Todas-Arestas exige que sejam criados casos de teste que passem ao menos uma vez pelas arestas (1, 2), (2, 2) e (2, 3).
- **Todos-Caminhos:** requer que todos os caminhos possíveis do programa sejam executados.
- **Exemplo:** com base na Figura 3, o critério Todos-Caminhos exige que sejam definidos casos de teste que percorram os caminhos (1, 2, 2, 3) ao menos uma vez.

De acordo com os critérios Todos-Nós, Todas-Arestas e Todos-Caminhos, caso não seja possível a criação de um caso de teste que alcance um nó, uma aresta ou um possível caminho do programa, um defeito potencial foi identificado.

TEMA 4 – TESTE FUNCIONAL

Ao contrário do teste estrutural em que o código fonte do programa é conhecido, e por isso é identificado como teste de caixa branca, os testes funcionais são executados sobre as entradas e saídas do programa sem que se tenha conhecimento do seu código-fonte, sendo, portanto, identificados como testes de caixa preta.

Conforme Delamaro, Maldonado e Jino (2007), o teste funcional pode detectar todos os defeitos, submetendo o programa a todas as possíveis entradas, o que é denominado *teste exaustivo*. Porém, como abordamos no Tema 1 desta aula ao conceituarmos dados e casos de teste, dada a cardinalidade do domínio de entrada de um programa, pode ser impraticável testá-lo com esse domínio, sendo necessário identificar subconjuntos de casos de teste.

Nesse contexto, podemos destacar como principais critérios do teste funcional o particionamento em classes de equivalência, a análise do valor-limite e o *error-guessing*.

4.1 PARTICIONAMENTO EM CLASSES DE EQUIVALÊNCIA

Considerando que o teste exaustivo é, em geral, impossível de se aplicar, torna-se necessária a identificação de subconjuntos de casos de teste (Delamaro; Maldonado; Jino, 2007). Com isso, parafraseando Wazlawick (2013), se um programa aceita um conjunto de dados e rejeita outro conjunto, pode-se dizer que existem duas classes de equivalência para os dados de entrada do programa.

Porém, como vimos anteriormente, esse conjunto de dados aceitos pelo programa corresponde ao seu domínio de entrada, podendo ser impossível testar todos os elementos de cada conjunto. Assim, segundo Wazlawick (2013), o particionamento de equivalência vai determinar que pelo menos um elemento de cada conjunto seja testado.

Conforme abordado por Myers et al. (2004), a técnica de particionamento de equivalência considera a divisão das entradas de um programa da seguinte forma:

1. Se as entradas válidas são especificadas como um intervalo de valores (por exemplo, de 10 a 20), então é definido 1 conjunto válido (de 10 a 20) e 2 inválidos (menor do que 10 e maior do que 20).

2. Se as entradas válidas são especificadas como uma quantidade de valores (por exemplo, uma lista com 5 elementos), então é definido 1 conjunto válido (lista com 5 elementos) e 2

inválidos (lista com menos de 5 elementos e lista com mais de 5 elementos).

3. Se as entradas válidas são especificadas como um conjunto de valores aceitáveis que podem ser tratados de forma diferente (por exemplo, as *strings* "masculino" e "feminino"), então é definido 1 conjunto válido para cada uma das formas de tratamento e 1 conjunto inválido para outros valores quaisquer.

4. Se as entradas válidas são especificadas como uma condição do tipo "deve ser de tal forma" (por exemplo, uma restrição sobre os dados de entrada como "a data final deve ser posterior à data inicial"), então devem ser definidos 1 conjunto válido (quando a condição é verdadeira) e 1 inválido (quando a condição é falsa).

Exemplificando com o nosso famoso programa que computa x^y, o domínio de entrada desse programa corresponde ao grupo 1, no qual as entradas são especificadas como um intervalo de valores. Em nosso caso, esse intervalo de valores é a partir do número 2 até o infinito para números inteiros. Portanto, poderíamos definir os seguintes conjuntos de casos de teste:

- **Conjunto válido:**
- ((2, 3), 8), ((3, 2), 9), ((5, 6), 15625), ((4, 3), 64)
- **Conjunto inválido 1:**
- ((2, 1), "Erro"), ((3, 0), "Erro"), ((5, -2), "Erro"), ((4, 1), "Erro")
- **Conjunto inválido 2:**
    - ((1, 3), "Erro"), ((-5, 2), "Erro"), ((0, 6), "Erro"), ((-4, 3), "Erro")

4.2 ANÁLISE DE VALOR-LIMITE

Ao contrário do particionamento em classes de equivalência, a análise de valor-limite, conforme colocado por Wazlawick (2013), consiste em considerar não apenas um valor qualquer para teste dentro de uma classe de equivalência, mas um ou mais valores fronteiriços com outras classes de equivalência quando isso puder ser determinado.

Com relação a determinar esses valores fronteiriços, Delamaro, Maldonado e Jino (2007) apresentam as seguintes diretrizes:

1. Se a condição de entrada especifica um intervalo de valores, devem ser definidos dados de teste para os limites desse intervalo e dados de teste imediatamente subsequentes, que

explorem as classes inválidas vizinhas desses intervalos.

2. Se a condição de entrada especifica uma quantidade de valores, por exemplo, de 1 a 255 valores, devem ser definidos dados de teste com nenhum valor de entrada, somente um valor, 255 valores e 256 valores de entrada.

3. Usar a diretriz 1 para as condições de saída.

4. Usar a diretriz 2 para as condições de saída.

5. Se a entrada ou saída for um conjunto ordenado, deve ser dada maior atenção ao primeiro e ao último elementos desse conjunto.

6. Usar a intuição para definir outras condições-limites.

Tendo como base essas seis diretrizes para definição dos conjuntos de casos de teste e sabendo que temos somente como valor-limite de entrada do nosso programa que computa x^y números inteiros maiores que 1, poderíamos definir os seguintes conjuntos de casos de teste de acordo com a diretriz 1:

- **Conjunto de casos de teste no limite:**
- ((2, 2), 4)
- **Conjunto de casos de teste imediatamente inferior ao limite:**
- ((2, 1), “Erro”), ((1, 2), “Erro”), ((1, 1), “Erro”),
- **Conjunto de casos de teste imediatamente superior ao limite:**
- Impossível determinar.

4.3 ERROR-GUESSING

Podemos considerar essa técnica como uma das mais utilizadas pelos próprios programadores durante o desenvolvimento do programa. Conforme destaca Delamaro, Maldonado e Jino (2007, p. 24): “Essa técnica corresponde a uma abordagem ad-hoc na qual a pessoa pratica, inconscientemente, uma técnica para projeto de casos de teste, supondo por intuição e experiência alguns tipos prováveis de erro e, a partir disso, definem-se casos de teste que poderiam detectá-los.”

Ainda segundo Delamaro, Maldonado e Jino (2007), esse critério objetiva enumerar possíveis erros ou situações propensas a erros e então definir casos de teste para explorá-los.

TEMA 5 – TESTE BASEADO EM DEFEITOS

Na técnica de teste baseado em defeitos, conforme destaca Delamaro, Maldonado e Jino (2007), são utilizados defeitos típicos do processo de implementação de *software* para que sejam derivados os requisitos de teste. Nessa técnica, o critério teste de mutação, que veremos na sequência, ou análise de mutantes, como também é conhecido, destaca-se como um dos critérios empiricamente mais eficazes na detecção de defeitos, conforme abordagens de DeMillo, Lipton e Sayward (1978) e Budd (1980).

5.1 TESTE DE MUTAÇÃO

Conforme abordagem de Delamaro, Maldonado e Jino (2007), o teste de mutação consiste em realizar diversas alterações no programa que está sendo testado, em que cada alteração simula um possível defeito no sistema, criando um conjunto de programas alternativos denominados *mutantes*. Cada alteração produzida no programa dá origem a um novo programa mutante.

Nesse contexto, conforme destacado por Delamaro, Maldonado e Jino (2007), para aplicar o teste de mutação, dado um programa **P** e um conjunto de casos de teste **T**, cuja qualidade se deseja avaliar, os seguintes passos para aplicar o critério são estabelecidos:

1. Geração dos mutantes.

2. Execução do programa em teste.

3. Execução dos mutantes.

4. Análise dos mutantes vivos.

Na sequência, abordaremos separadamente cada um dos passos da aplicação do teste de mutação, com uma aplicação de exemplo em nosso programa que computa x^y.

5.1.1 GERAÇÃO DOS MUTANTES

Conforme abordamos anteriormente, um mutante é uma cópia do programa sendo testado com uma pequena alteração no código-fonte em relação ao código do programa original.

Para gerar o conjunto de mutantes, com todas as possibilidades de simulações de erro no código original, são utilizados operadores de mutação. Deste modo, de acordo com Delamaro, Maldonado e Jino (2007), um operador de mutação aplicado a um programa **P** transforma **P** em um programa similar, ou seja, um mutante.

Porém, conforme destaca Delamaro, Maldonado e Jino (2007), além dos tipos de defeitos que se desejam revelar e da cobertura que se quer garantir, a escolha de um conjunto de operadores de mutação depende também da linguagem de programação em que o programa em teste foi escrito. Delamaro et al. (2007, p. 85) destacam a seguinte relação apresentada na Tabela 1.

Tabela 1 – Operadores de mutação por linguagens de programação

Linguagem de programação	Operadores de mutação
Fortran	22 operadores
Cobol	27 operadores
C	77 operadores

Fonte: elaborado com base em Delamaro; Maldonado; Jino, 2007.

Como exemplo de um operador de mutação, vamos considerar os operadores de mutação para a linguagem de programação C apresentados por Delamaro, Maldonado e Jino (2007) na Tabela 2.

Tabela 2 – Exemplo de operadores de mutação para a linguagem C

Operador de mutação	Tipo de mutação
SSDL	Eliminação de comandos
ORRN	Troca de operador relacional
STRI	Armadilha em condição de comando *if*
Vsrr	Troca de variáveis escalares

Fonte: elaborado com base em Delamaro; Maldonado; Jino, 2007.

Para exemplificar a aplicação desses operadores de mutação, vamos aplicá-los ao nosso programa que computa x^y, conforme vemos na Tabela 3.

Tabela 3 – Geração dos mutantes do programa que computa x^y.

Mutante SSDL	Mutante ORRN	Mutante Vsrr

enquanto ((x <= 1) ou (y <= 1)) {
escreva("Valores incorretos...")
escerva("Digite o valor da...")
leia(x, y)
}
	enquanto ((x **>**= 1) ou (y <= 1)) {
	escreva("Valores incorretos...")
	escerva("Digite o valor da...")
	leia(x, y)
	}
		enquanto ((**y** <= 1) ou (y <= 1)) {
		escreva("Valores incorretos...")
		escerva("Digite o valor da...")
		leia(x, y)
		}

result = x	enquanto ((x <= 1) ou (y **>**= 1)) {
	escreva("Valores incorretos...")
	escerva("Digite o valor da...")
	leia(x, y)
	}
		enquanto ((**y** <= 1) ou (**x** <= 1)) {
		escreva("Valores incorretos...")
		escerva("Digite o valor da...")
		leia(x, y)
		}

Como podemos observar na Tabela 3, cada um dos destaques em vermelho no código-fonte é uma alteração realizada pelo respectivo operador de mutação, gerando um novo programa chamado *mutante*. Vale destacar aqui que na Tabela 3 está sendo representada somente a parte do código-fonte alterada pelo operador mutação, pois o restante do código permanece igual ao original.

No caso do operador de mutação SSDL, serão gerados dois programas mutantes com o código-fonte destacado em vermelho removido, permanecendo ausente em relação ao programa original. Para o operador ORRN, serão gerados dois programas mutantes com os operadores lógicos destacados em vermelho alterados em relação ao programa original. Já para o operador Vsrr, serão gerados dois programas mutantes com as variáveis destacadas em vermelho substituídas em relação ao programa original. Foram gerados somente dois mutantes para cada operador por motivos de exemplificação, pois devem ser gerados tantos quanto possível.

5.1.2 EXECUÇÃO DO PROGRAMA EM TESTE

Para a execução do programa em teste, deve ser gerado um conjunto de casos de teste **T**, conforme discutimos no Tema 1 desta aula. Supomos, que para o nosso programa que computa x^y, temos o seguinte conjunto de casos de teste:

$$\mathbf{T} = \{((1, 2), \text{“Erro”}), ((2, 3), 8), ((3, 2), 6)\}$$

Assim, essa etapa consiste em executar o programa em teste com **T** e verificar se o seu comportamento é o esperado. A Tabela 4 apresenta os resultados esperados e obtidos após a execução do programa que computa x^y com **T**.

Tabela 4 – Resultado da execução do programa que computa x^y com T

Casos de teste	Resultado esperado	Resultado obtido
(1, 2)	"Erro"	"Erro"
(2, 3)	8	8
(3, 2)	9	9

Conforme podemos ver na Tabela 4, nosso programa não apresentou nenhum erro ao ser executado com T, pois todos os resultados obtidos são os mesmos que os esperados. Isso não garante que nosso programa não contenha defeitos, pois, com base em Delamaro, Maldonado e Jino (2007), sabemos que não é possível, em geral, provar por meio de testes a correção de um programa. É preciso, então, mostrar até que ponto pode-se chegar com a aplicação de testes.

5.1.3 EXECUÇÃO DOS MUTANTES

Esta etapa consiste em executar todos os programas mutantes gerados pelos operadores de mutação com o mesmo conjunto **T** aplicado no programa original. Para cada dado de teste aplicado em um mutante, o resultado obtido da execução do programa mutante é comparado com o resultado obtido na execução do programa original.

Toda vez que o resultado do mutante difere do programa original com algum caso de teste de **T**, esse mutante é considerado morto e descartado. Para um mutante que apresenta o mesmo resultado do programa original, sem que seja possível gerar novos dados de teste que identifiquem a alteração gerada pelo operador de mutação, um defeito pode ter sido revelado ou esse mutante pode ser considerado equivalente ao programa original.

Tabela 5 – Execução dos mutantes com T

Dados de teste	Mutante	Resultado programa original	Resultado mutante
(1, 2)	enquanto ((x <= 1) ou (y <= 1)) { escreva("Valores incorretos...")	"Erro"	1

	escerva(“Digite o valor da...”) leia(x, y) }		
(2, 3)	result = x	8	“Erro”

(2, 3)	enquanto ((x **>**= 1) ou (y <= 1)) {
	escreva(“Valores incorretos...”)
	escerva(“Digite o valor da...”)
	leia(x, y)
	}
		8	“Erro”

(2, 3)	enquanto ((x <= 1) ou (y **>**= 1)) {
	escreva(“Valores incorretos...”)
	escerva(“Digite o valor da...”)
	leia(x, y)
	}
		8	“Erro”

(1, 2)	enquanto ((**y** <= 1) ou (y <= 1)) {
	escreva(“Valores incorretos...”)
	escerva(“Digite o valor da...”)
	leia(x, y)
	}
		“Erro”	1

(2, 3)	enquanto ((**y** <= 1) ou (**x** <= 1)) {
	escreva(“Valores incorretos...”)
	escerva(“Digite o valor da...”)
	leia(x, y)
	}
		8	8

De acordo com a Tabela 5, somente um mutante apresentou resultado igual ao programa original, sem ser possível a geração de um dado de teste que diferencie o resultado da execução desse mutante com o resultado da execução do programa original. Portanto, neste exemplo, de um total de 6 mutantes gerados, obtivemos 5 mutantes que apresentaram resultado diferente com algum caso de teste, sendo, portanto, considerados mortos e descartados e somente um igual ao original que é considerado vivo para análise posterior.

Com o objetivo de averiguar a adequação dos casos de teste utilizados, é aplicado o escore de mutação. De acordo com Delamaro, Maldonado e Jino (2007), o escore de mutação varia entre 0 e 1 e fornece uma medida objetiva de quanto o conjunto de casos de teste analisado aproxima-se da

adequação. Assim, dado um programa **P** e o conjunto de casos de teste **T**, calcula-se o escore de mutação *ms(P, T)* da seguinte maneira:

$$ms(P,T) = \frac{DM(P,T)}{M(P) - EM(P)}$$

Em que:

- **DM(P, T):** número de mutantes mortos pelo conjunto de casos de teste.
- **M(P):** número total de mutantes gerados com base no programa P.
- **EM(P):** número de mutantes gerados que são equivalentes.

Para efeito de exemplificação, consideraremos que o mutante que apresentou resultado igual ao programa original é equivalente, ou seja, está correto, então obtemos o seguinte escore de mutação para o nosso teste:

$$ms(P,T) = \frac{5}{6-1} = 1$$

Por outro lado, caso o mutante que consideramos como equivalente no exemplo anterior seja um defeito, obteremos o seguinte escore de mutação:

$$ms(P,T) = \frac{5}{6-0} = 0{,}83$$

5.1.4 ANÁLISE DOS MUTANTES VIVOS

Esta etapa aplica-se para identificar, entre todos os mutantes que permaneceram vivos, quais representam um defeito no programa original ou quais são equivalentes.

Em nosso exemplo, o mutante é equivalente ao progrma original, pois a mutação realizada apenas inverte a posição das variáveis *x* e *y* na comparação se os seus valores são menores ou iguais a 1. Portanto, a ordem de posicionamento dessas variáveis não influencia logicamente o resultado do programa, sendo deste modo um mutante equivalente ao programa original.

FINALIZANDO

Nesta aula, estudamos as principais técnicas de teste de *software* e seus critérios. Pudemos perceber que existem várias maneiras de testar o *software*, desde a visão do programador, com os

testes específicos no código-fonte do programa, conhecidos como testes de caixa branca, até os testes de caixa preta, que variam desde a visão do programador até a visão do usuário.

Aprendemos também que não se prova a ausência de defeitos em um programa por meio do teste. É impossível provar por qualquer técnica de teste que não existam defeitos no programa, pois, conforme relatam Myers e Sandler (2004), a atividade de teste é o processo de executar um programa com a intenção de revelar defeitos, e não provar a ausência deles.

Encerramos a aula com a técnica de teste baseada em defeitos, na qual estudamos o critério de teste análise de mutantes, que resumidamente consiste em simular defeitos conhecidos no código-fonte e confrontá-los com o código original, objetivando revelar esses defeitos no programa em teste e avaliar a qualidade dos casos de teste utilizados.

Com esta aula de testes, finalizamos uma sequência básica do desenvolvimento de *software*, que consiste na seleção de processo de desenvolvimento, metodologia ágil, projeto do *software*, definição da arquitetura e testes. Portanto, na próxima aula, abordaremos os conceitos do DevOps, com o objetivo de conhecer um conjunto de práticas para integração entre as equipes de desenvolvimento de *softwares*.

REFERÊNCIAS

BUDD, T. A. **Mutation Analysis of Program Test Data**. Tese (Doutorado) – Yale University, Yale,1980.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao teste de software**. Rio de Janeiro: Elsevier, 2007.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. **Computer**, v. 11, n.4, p. 34-41, 1978.

IEEE 610-1990 – **Dicionário de computador padrão IEEE**: uma compilação de glossários de computador padrão IEEE. [S.l.]: IEEE Computer Society, 1990.

MYERS, G. J. et al. **The Art of Software Testing**. 2. ed. New Jersey: John Wiley & Sons, 2004.

MYERS, G. J.; SANDLER, C. **TheArt of Software Testing**. [S.l.]: John Wiley & Sons, 2004.

PRESSMAN, R. S. **Engenharia de software:** uma abordagem profissional. 7. ed. Porto Alegre: AMGH, 2011.

WAZLAWICK, R. S. **Engenharia de software:** conceitos e práticas. São Paulo: Elsevier, 2013.